

# 漫画：Bitmap算法 整合版

Original:玻璃猫 程序员小灰 2017-08-28

小灰，最近在忙什么呢？



哎，最近在给项目做性能优化，  
头疼啊.....



什么样的项目？出了什么性能  
问题啊？



哎，这事说来话长.....



两个月之前——

产品经理：小灰，为了帮助公司精准定位用户群体，咱们需要开发一个用户画像系统，实现用户信息的标签化。



用户标签包含用户的社会属性、生活习惯、消费行为等等主要信息，比如下面这样子：



## 小灰的用户标签



通过用户标签，我们可以实现多样的用户群体统计。比如统计用户的男女比例，统计喜欢旅游的用户数量等等。



放心吧，这个需求交给我妥妥的！



为满足用户标签的统计需求，小灰利用Mysql设计了如下的表结构，每一个维度的标签都对应着Mysql表的一列：

Name	Sex	Age	Occupation	Phone
小灰	男	90后	程序员	苹果
大黄	男	90后	程序员	三星
小白	女	00后	学生	小米

要想统计所有90后的程序员该怎么做呢？

用一条求交集的SQL语句即可：

```
Select count ( distinct Name ) as 用户数 from table where age = '90后' and Occupation = '程序员' ;
```

要想统计所有使用苹果手机或者00后的用户总合该怎么做？

用一条求并集的SQL语句即可：

```
Select count ( distinct Name ) as 用户数 from table where Phone = '苹果' or age = '00后' ;
```

看起来很简单嘛，嘿嘿.....



两个月之后——

事情没那么简单，标签越来越多，都快上干了，这得要多少列啊……



筛选标签过多的时候，拼接出来的  
SQL语句像面条一样长……



多个用户群体求并集的时候要做  
distinct，性能实在太慢了……



天呐，这可怎么办……



---

事情就是这样，愁死我了……



哈哈，小灰，你听说过 Bitmap  
算法吗？中文又叫做位图算法。



我又不是搞计算机图形学的，  
研究位图算法干嘛？



这里所说的位图不是像素图片，而是内存中连续的二进制位 (bit)，用于对大量整型数据做去重和查询。



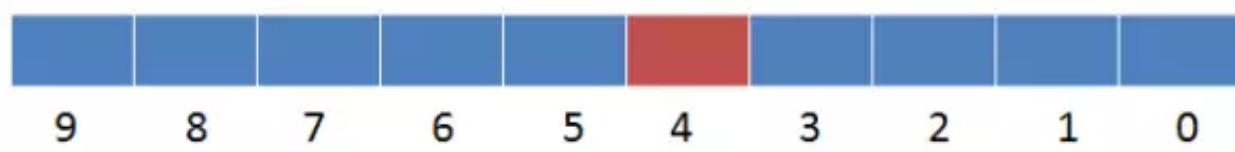
举个例子，给定一块长度是 10bit 的内存空间，想要依次插入整型数据 4, 2, 1, 3，我们需要怎么做呢？



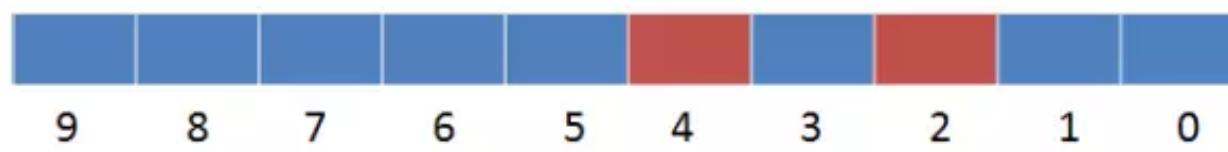
1. 给定长度是10的bitmap，每一个bit位分别对应着从0到9的10个整型数。此时bitmap的所有位都是0。



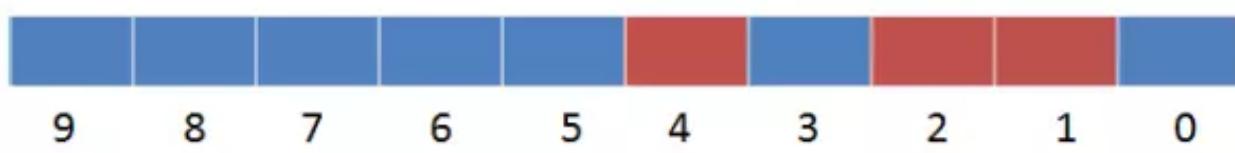
2. 把整型数4存入bitmap，对应存储的位置就是下标为4的位置，将此bit置为1。



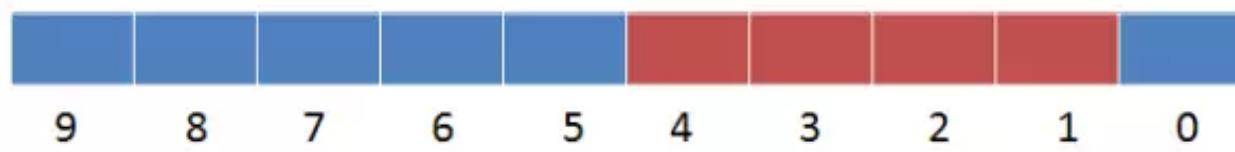
3. 把整型数2存入bitmap，对应存储的位置就是下标为2的位置，将此bit置为1。



4. 把整型数1存入bitmap，对应存储的位置就是下标为1的位置，将此bit置为1。



5. 把整型数3存入bitmap，对应存储的位置就是下标为3的位置，将此bit置为1。



要问此时bitmap里存储了哪些元素？显然是4,3,2,1，一目了然。

Bitmap不仅方便查询，还可以去除掉重复的整型数。

看起来有点意思，可是 bitmap 算法  
跟我的项目有什么关系呢？



你仔细想一想，你所做的用户标签  
能不能用 bitmap 的形式来存储呢？



我的每一条用户数据都对应着几百  
上千个标签，怎么也无法转换成  
Bitmap 的形式啊.....



别急，我们不妨把思路逆转一下，  
为什么一定要让一个用户对应多个  
标签，而不是一个标签对应一个用  
户呢？



一个标签对应多个用户？

让我想想啊.....



我明白了！我可以先建立一个用户名和用户 ID 的映射，然后让每一个标签存储包含此标签的所有用户的 ID，就像倒排索引一样！



1. 建立用户名和用户ID的映射：

Name	Sex	Age	Occupation	Phone
小灰	男	90后	程序员	苹果
大黄	男	90后	程序员	三星
小白	女	00后	学生	小米



ID	Name
1	小灰
2	大黄
3	小白

2. 让每一个标签存储包含此标签的所有用户ID，每一个标签都是一个独立的Bitmap。

ID	Sex	Age	Occupation	Phone
1	男	90后	程序员	苹果
2	男	90后	程序员	三星
3	女	00后	学生	小米



Sex	Bitmap
男	1, 2
女	3

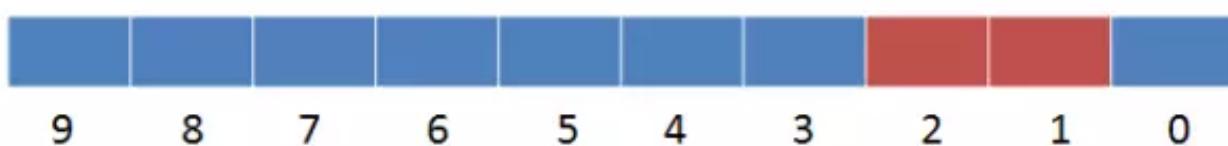
Age	Bitmap
90后	1, 2
00后	3

Occupation	Bitmap
程序员	1, 2
学生	3

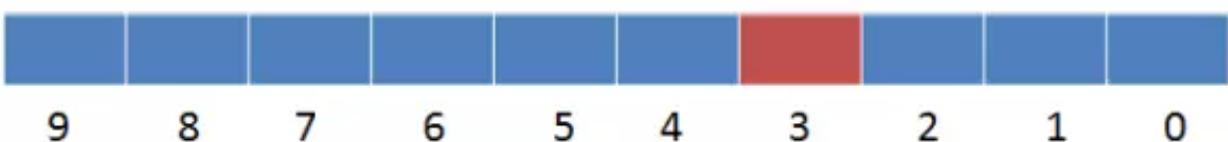
Phone	Bitmap
苹果	1
三星	2
小米	3

3. 这样，实现用户的去重和查询统计，就变得一目了然：

程序员：



00后：

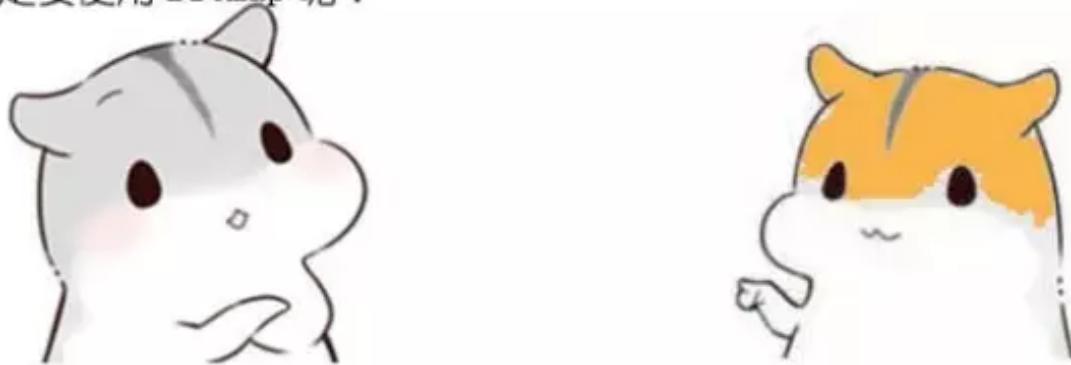


Bingo !

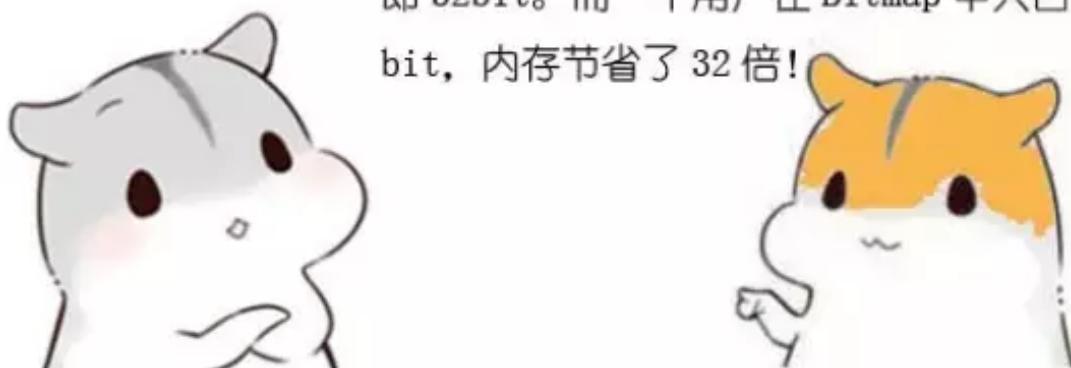
这就是 Bitmap 算法的运用。



我还有一点不明白，`HashSet` 和 `HashMap` 也同样能实现用户的去重和统计，为什么一定要使用 `Bitmap` 呢？



傻孩子，如果用 HashSet 或 Hashmap 存储的话，每一个用户 ID 都要存成 int，占 4 字节即 32bit。而一个用户在 Bitmap 中只占一个 bit，内存节省了 32 倍！



不仅如此，Bitmap 在用户群做交集和并集运算的时候也有极大的便利。我们来看看下面的例子：



## 1. 如何查找使用苹果手机的程序员用户？

程序员用户 (0000000110B) :



使用苹果手机的用户（0000000010B）：

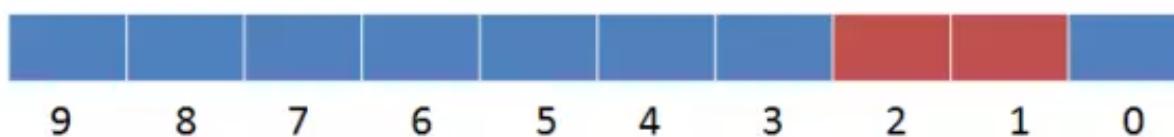


使用苹果手机的程序员用户 ( $0000000110B \& 0000000010B = 0000000010B$ ) :

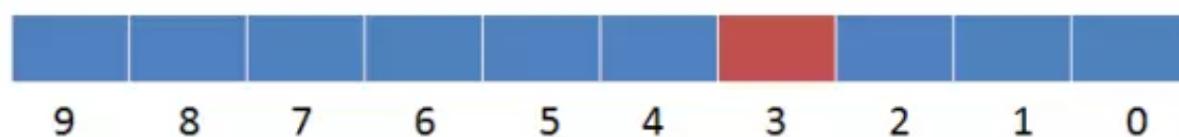


2. 如何查找所有男性或者00后的用户？

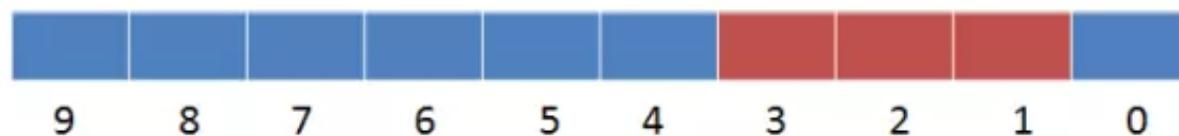
男性用户（0000000110B）：



00后用户（0000001000B）：



男性或00后用户（0000000110B | 0000001000B = 0000001110B）：



这就是 Bitmap 算法的另一个优势：  
位运算的高性能。



原来如此。Bitmap 这种解决方案  
这么方便，不知有没有什么缺点？



缺点也是存在的，Bitmap 不支持  
[ 非运算 ]。比如想要查找不使用  
苹果手机的用户，Bitmap 就无能  
为力了。



Bitmap 为什么不支持非运算呀？



嘿嘿，自己动动脑子就想明白啦。



呃，再问最后一个问题，现在有什  
么开源的 Bitmap 实现吗？



JDK 中的 BitSet 集合是对 Bitmap 算法相对简单的实现。而谷歌开发的 EWAHCompressedBitmap 则是一种更为优化的实现。



太好啦，我要把 Bitmap 的思想马上应用到项目当中。妈妈再也不用担心我的程序性能了！



一周之后.....

大黄，上次你科普了 Bitmap 算法，让我收获不小。不过我还是有很多疑问。



哈哈，有什么疑问尽管提出来吧，我知无不言，言无不尽。



首先，你上次说 Bitmap 的一个缺点  
是无法进行 [ 非运算 ]。我没想明白，  
为什么不能做非运算呢？



的确，在统计用户标签这样的特定场  
景下，Bitmap 无法 [ 直接 ] 做非运算。  
为什么呢？看看下面的例子：



我们以上一期的用户数据为例，用户基本信息如下。按照年龄标签，可以划分成90后、00后两个Bitmap：

ID	Sex	Age	Occupation	Phone
1	男	90后	程序员	苹果
2	男	90后	程序员	三星
3	女	00后	学生	小米



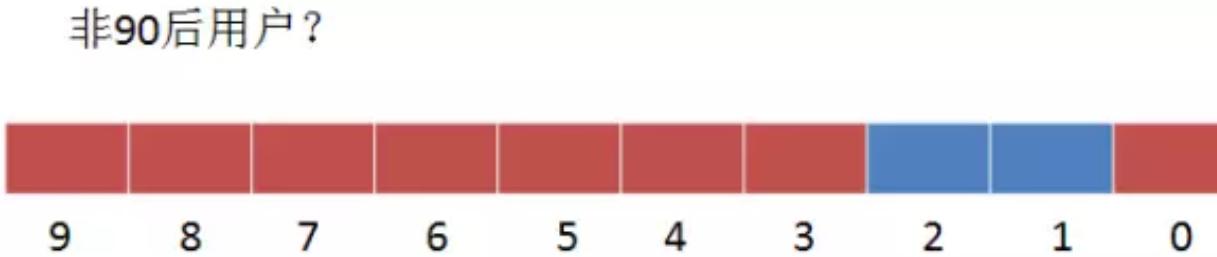
Age	Bitmap
90后	1, 2
00后	3

用更加形象的表示，90后用户的Bitmap如下：

90后用户：



这时候可以直接求得**非**90后的用户吗？直接进行非运算？



显然，非90后用户实际上只有1个，而不是图中得到的8个结果，所以不能直接进行非运算。

原来如此。那如果我们一定要求出  
不属于某个标签的用户数量，该怎  
么做呢？



其实也不难，我们只需借助一个  
全量的 Bitmap。



同样是刚才的例子，我们给定90后用户的Bitmap，再给定一个全量用户的Bitmap。最终要求出的是存在于全量用户，但又不存在于90后用户的部分。

90后用户：



全量用户：



如何求出呢？我们可以使用**异或**操作，即相同位为0，不同位为1。

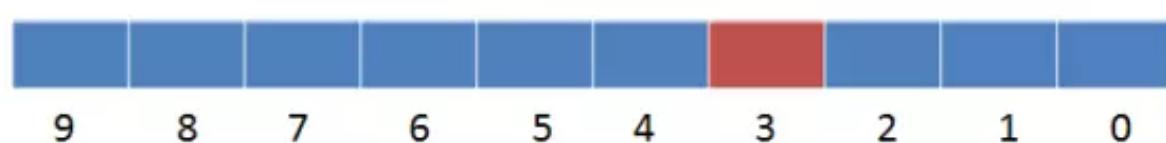
90后用户 (0000000110B) :



全量用户 (0000001110B) :



非90后用户 ( $0000000110B \text{ XOR } 0000001110B = 0000001000B$ ) :



我明白了，真是个好办法！



下面我还有一个疑问。如果在一个很长的  
Bitmap 里只存有一两用户，那样岂不是很  
浪费空间？



问得很好。在谷歌所实现的  
EWAHCompressedBitmap 中，对  
Bitmap 存储空间做了一定的优化。



在介绍具体优化过程之前，我们先  
要了解 Bitmap 的存储方式。EWAH  
把 Bitmap 存储于 long 数组当中。



long 数组的每一个元素都可以当做  
是 64 位的二进制数，也是整个  
Bitmap 的子集。谷歌把这些子集叫  
做 [Word]。



00001110B

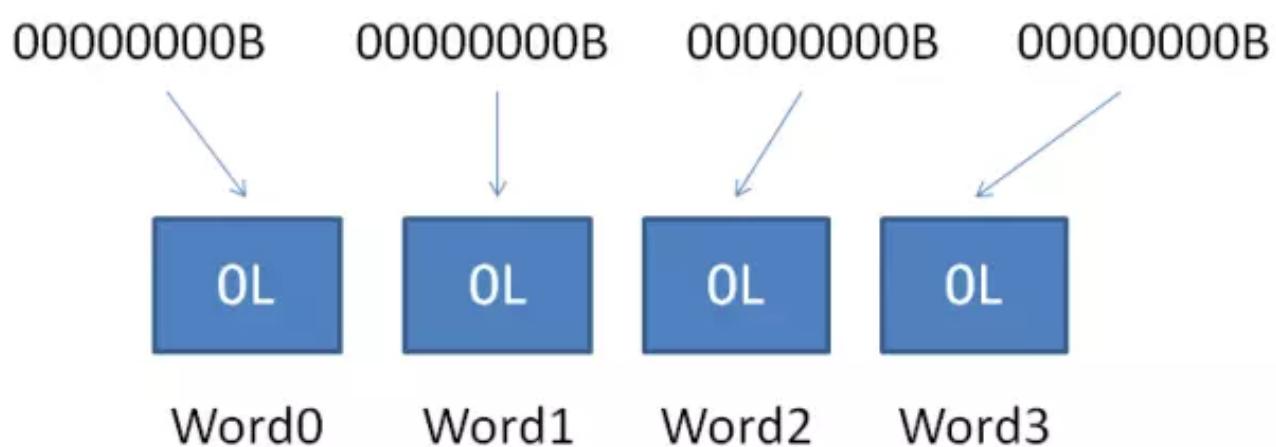
15L

Word

在一个 Bitmap 中，一共存在  
多少个 Word 呢？



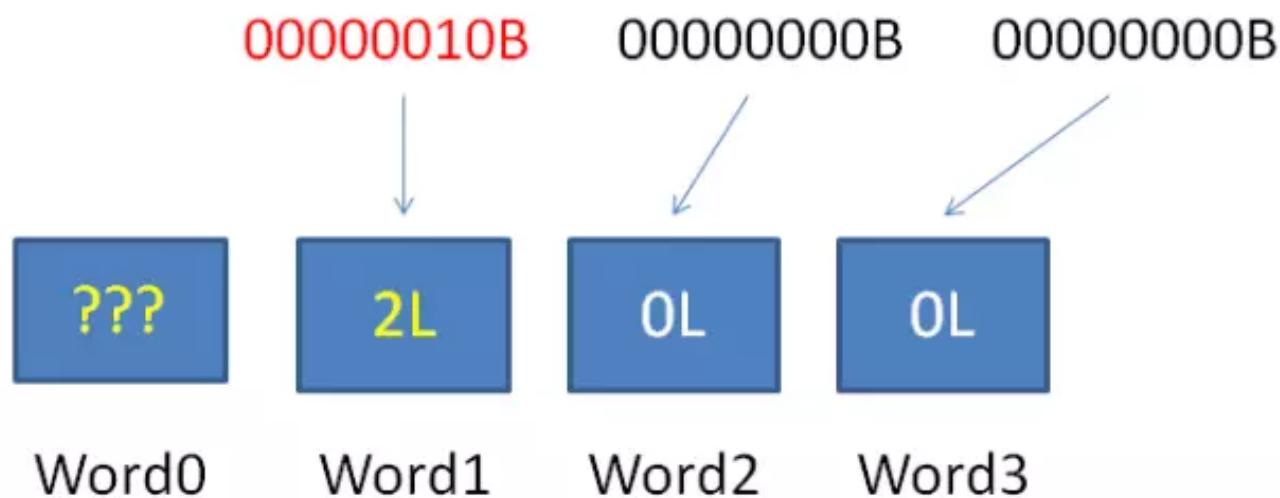
当创建一个空的 Bitmap 时，初始只  
有 4 个 Word，也即 long 数组的长度  
是 4。随着数据的不断插入，Word 数  
量会随之扩充。



由于还未插入任何数据，此时所有的  
Word 值都是 0。其中 word0 是不直接存  
储数据的，姑且可以认为它存储的是  
Bitmap 的头信息。



现在让我们来插入一个 ID 是 1 的用户。由于 Word0 不直接存数据，所以插入到 Word1 中，结果如下：



插入的用户 ID 是 1，为什么 word1  
的值变成了 2 ?

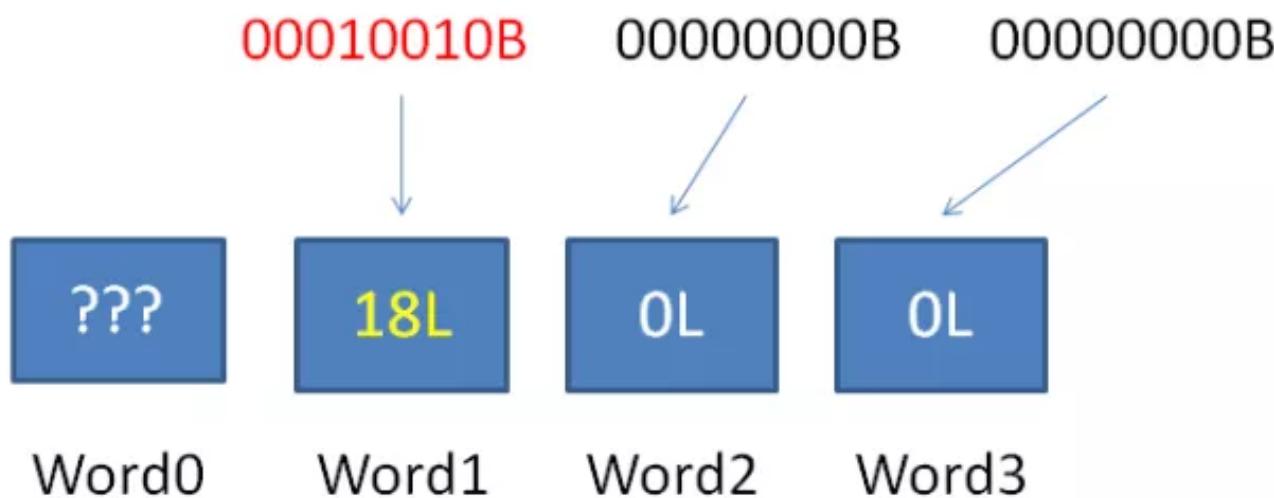
Word0 里面的问号又是什么意思？



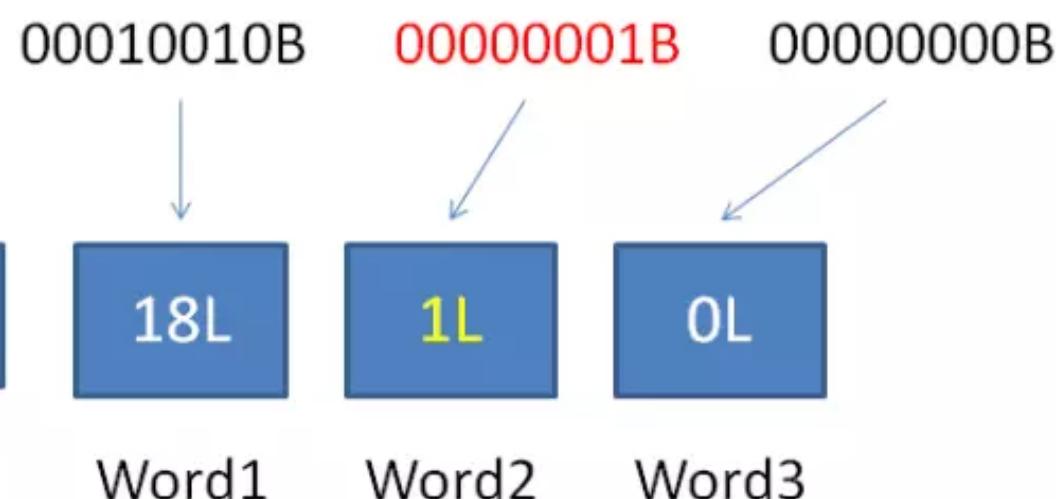
Word1 的结果之所以是 2，也即  
000000010B，是因为最右边的二进制  
位代表 ID 是 0 的用户。



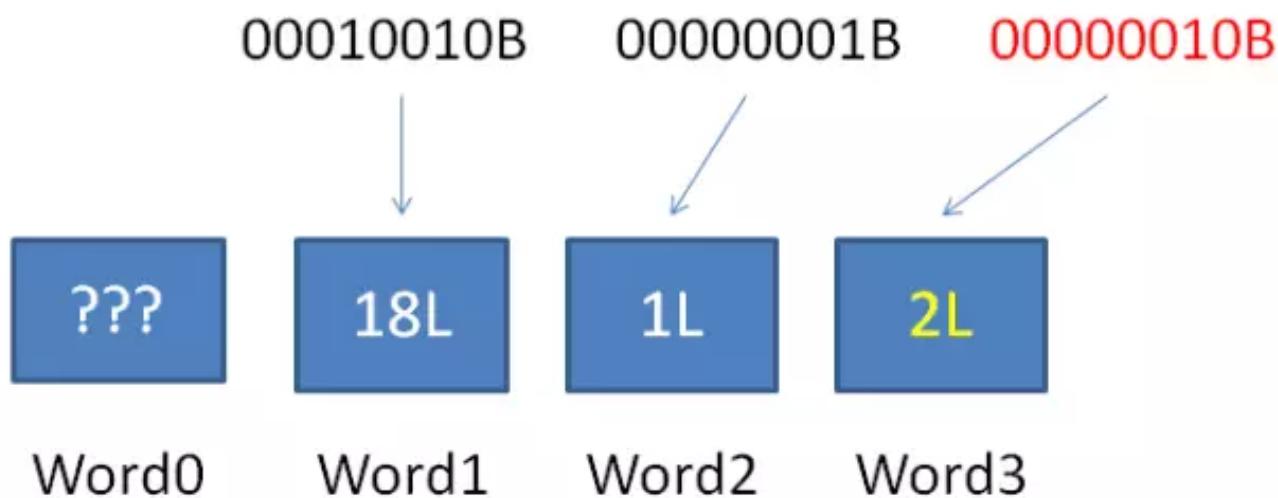
至于 Word0 当中的值变化，我们稍后再做解释。这时候我们再插入 ID 是 4 的用户，结果如下：



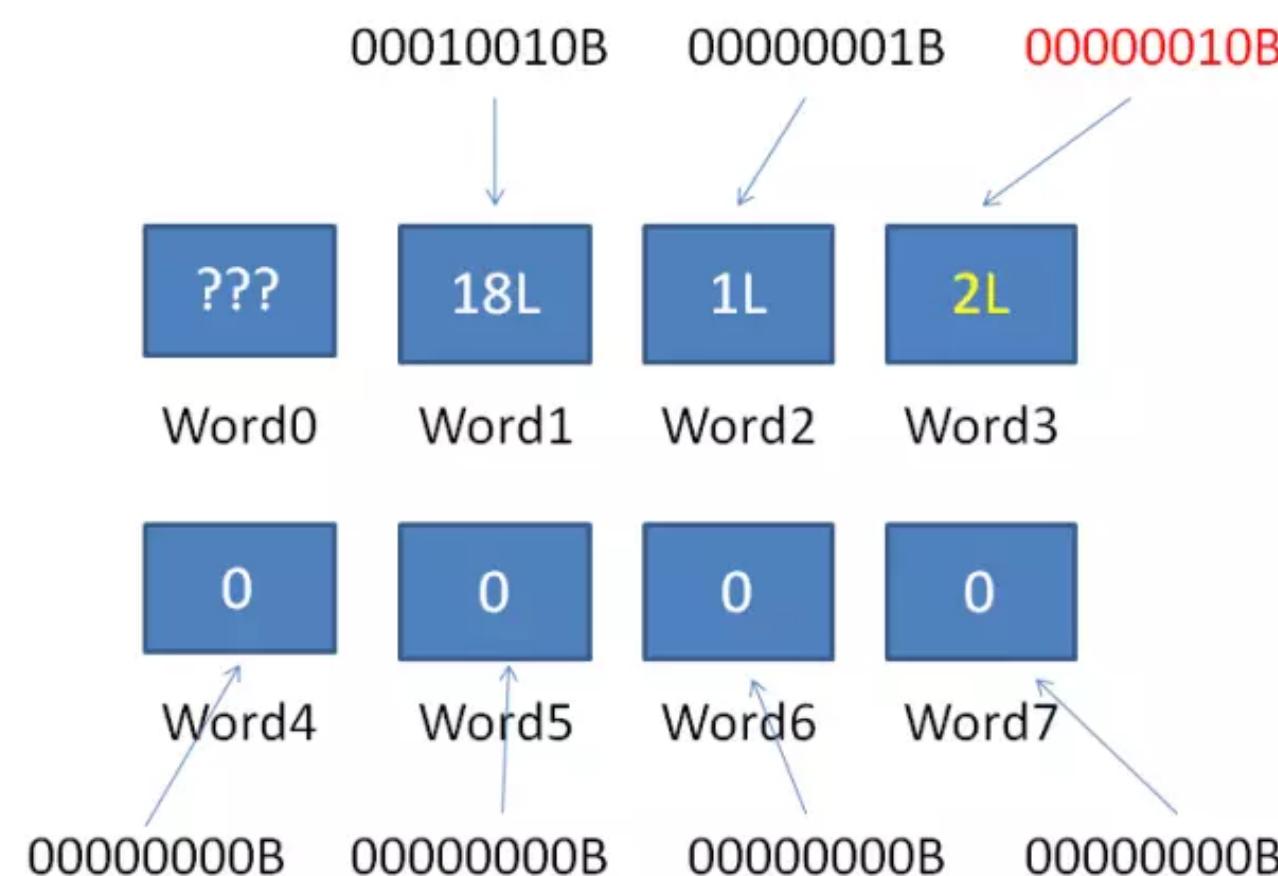
上面的结果显而易见。下面我们再来插入 ID 是 64 的新用户，由于 Word1 存储范围只有 0-63，因此 64 会存入 Word2。



下面再插入一个 ID 是 129 的用户。  
由于前两个 Word 最大可以存储到  
127，因此 129 应当存入 Word3。



这个时候所有的 Word 都已经被占  
用，Bitmap 将会进行动态扩容。  
扩容结果如下：



有个问题，如果要插入一个 ID 为  
400000 的用户怎么办？是不是要把这  
个数量之前的所有 Word 都创建出来？



问得很好，这一点正是 Bitmap 优化的  
所在。咱们姑且不管前面会有多少个  
Word，单看 400000 这个 ID 所在 Word，  
值是多少？



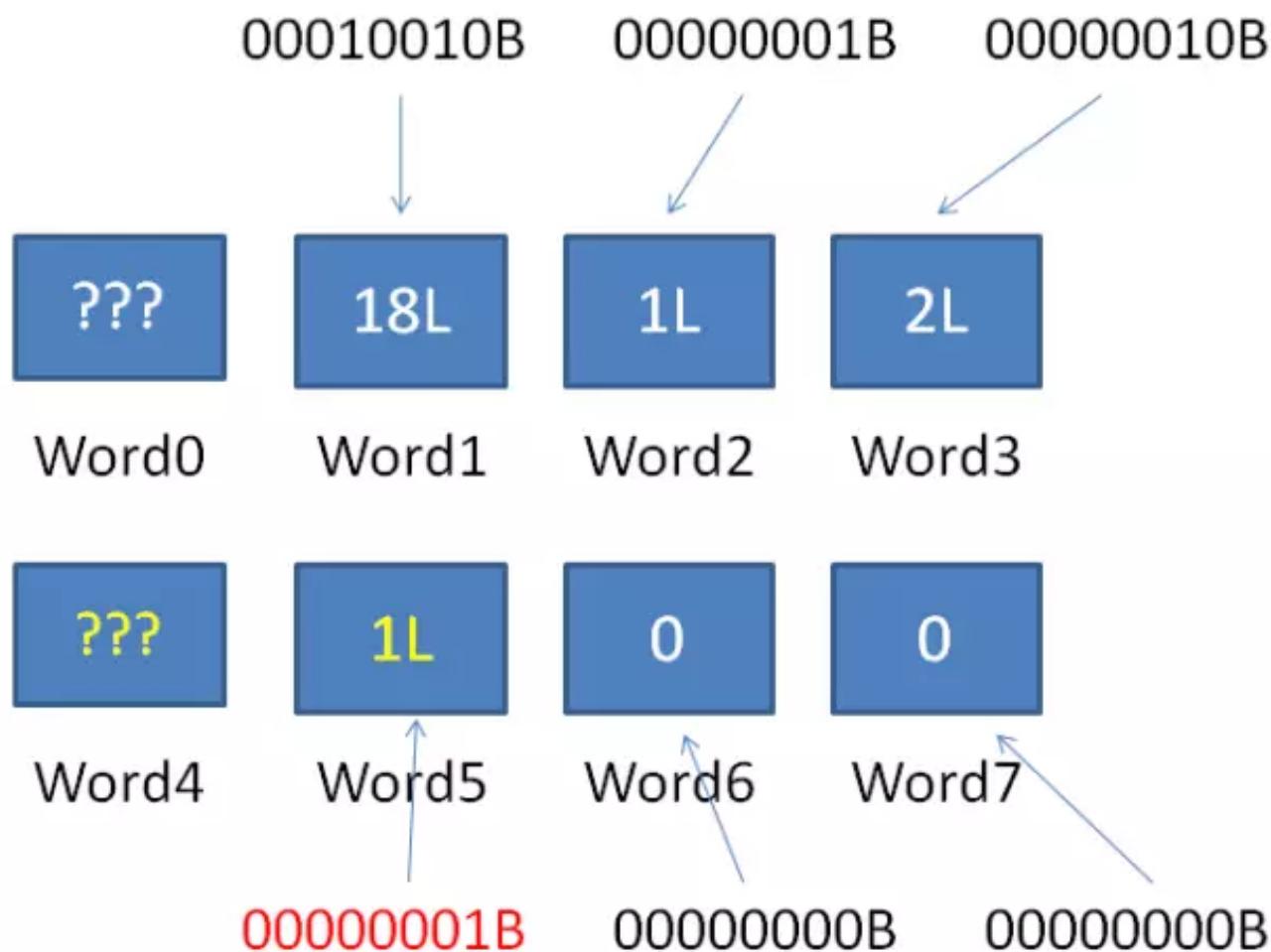
$(400000+1)$  除以 64 商 6250 余 1，  
那个 Word 的值应该是 1L 吧？



没错，这个 Word 本身的值是 1L，而  
刚才你也算出商是 6250，说明这个  
Word 前面应该有 6250 个 Word。



但是，Bitmap 不会傻傻地把长度扩充到 6000 多，只会改变后方两个 Word，Word4 设置成一个特殊 Word，Word5 成为存储了 400000 的 Word。



Word4 到底是什么鬼？明明横跨了将近 400000 个 ID，居然只额外使用了一个 Word ?



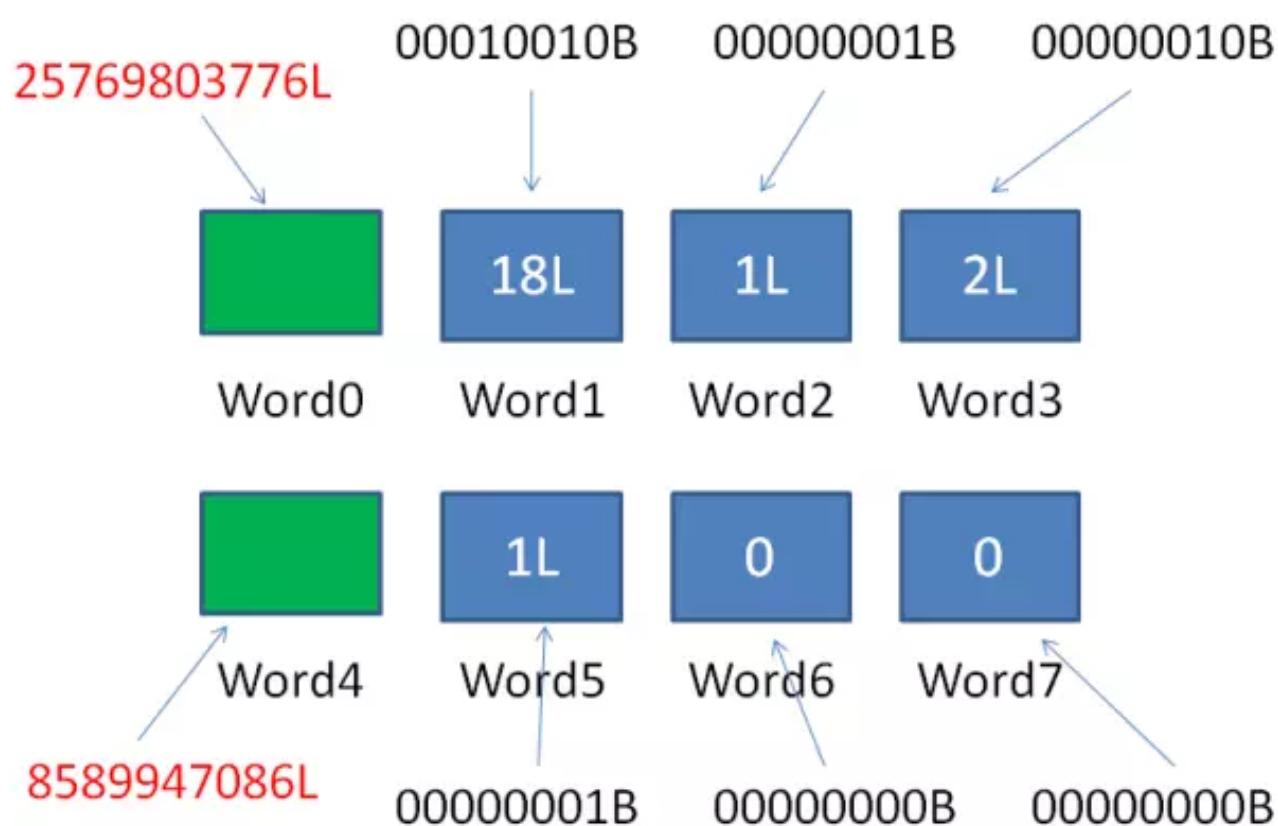
这里就引入了两个全新的概念：  
Running Length Word  
Literal Word



Word 分为两种，直接存储数据的叫 Literal Word，简称 LW。存储 [ 跨度信息 ] 的叫 Running Length Word，简称 RLW。



其中 Word0 和 Word4 就属于 RLW，它们的具体值是什么呢？我们来看一看。



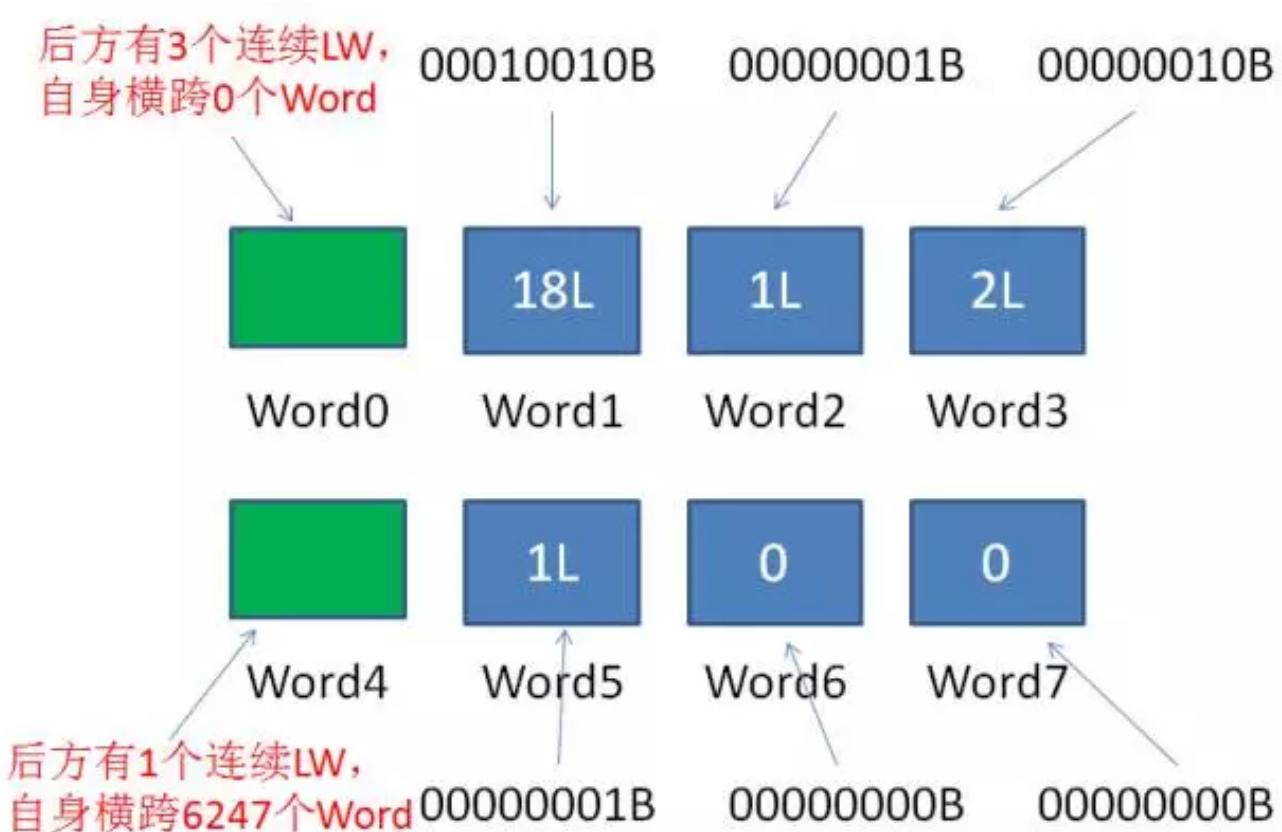
$$25769803776L = 11000000000000000000000000000000B$$

$$8589947086L = 1000000000000000000000000000000011000011001110B$$

哎妈呀，RLW 里面存储了这么长的  
数字，究竟是什么意思啊？



每一个 RLW 分成两部分，低 32 位表  
示当前 Word 横跨了多少个空 Word。  
高 32 位表示当前 RLW 后方有多少个  
连续的 LW。



对于极端稀疏的 Bitmap，这样果  
然节省了大量的空间，真是个聪  
明的办法！



还有一个疑问，既然 Bitmap 里有些 Word 是存储数据的，有些 Word 是跨度信息的，当一个新数据插入的时候，如何找到正确的位置呢？



这就要依靠每一个 RLW 作为 [ 路标 ]。比如我们要插入的新 ID 是 400003，寻找位置的过程如下：



1. 解析 Word0，得知当前 RLW 横跨的空 Word 数量为 0，后面有连续 3 个普通 Word。

2. 计算出当前 RLW 后方连续普通 Word 的最大 ID 是  $64 \times (0 + 3) - 1 = 191$ 。

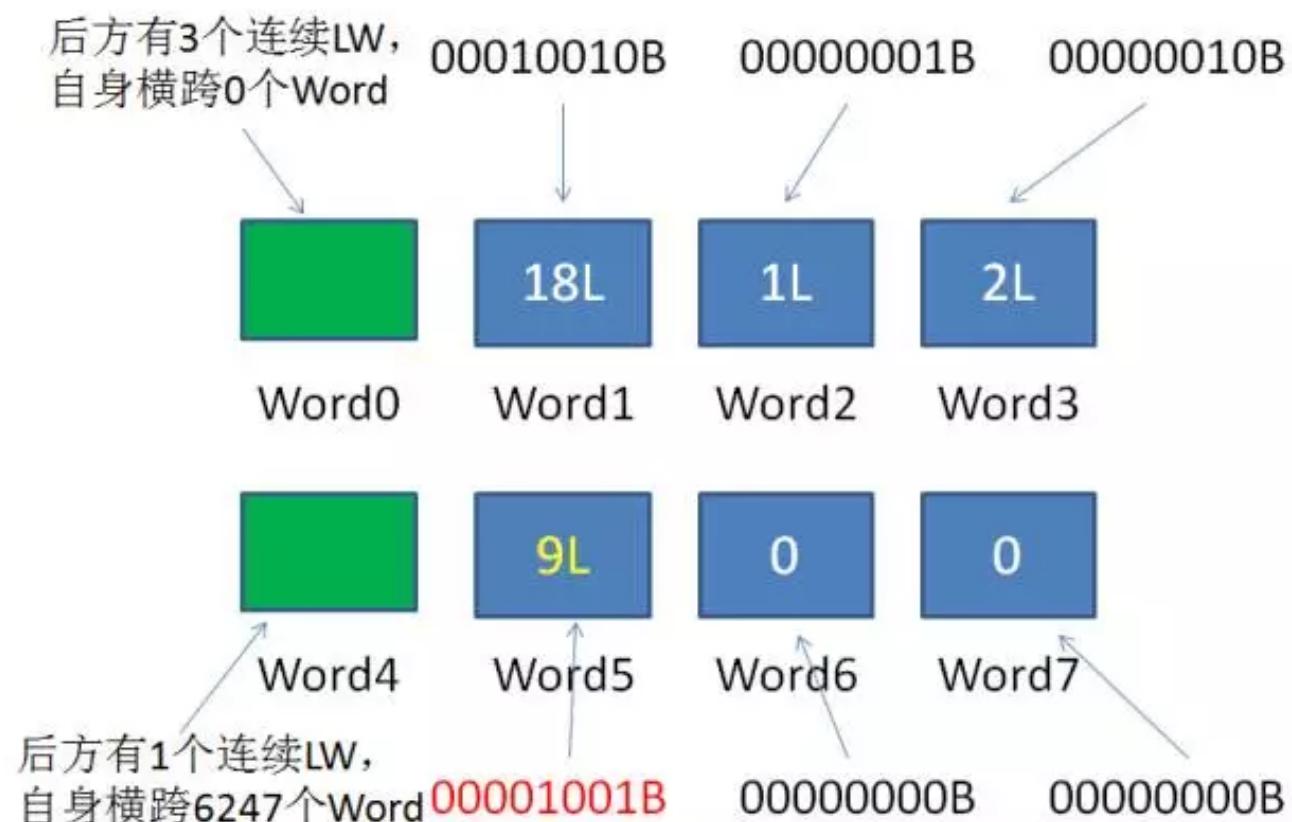
3. 由于  $191 < 400003$ ，所以新 ID 必然在下一个 RLW ( Word4 ) 之后。

4. 解析 Word4，得知当前 RLW 横跨的空 Word 数量为 6247，后面有连续 1 个普通 Word。

5. 计算出当前 RLW ( Word4 ) 后方连续普通 Word 的最大 ID 是  $191 + (6247 + 1) \times 64 = 400063$ 。

6. 由于  $400003 < 400063$ ，因此新 ID 400003 的正确位置就在当前 RLW ( Word4 ) 的后方普通 Word，也就是 Word5 当中。

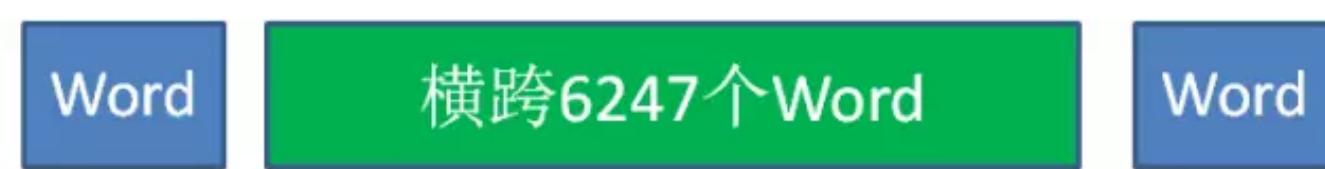
最终插入结果如下：



那么，如果新插入的 ID 刚好在存储横跨信息的 RLW 范围内，比如插入 200000，该怎么办呢？



这种情况，就会涉及原 RLW 的分裂，如果插入的 ID 是 200000，简化的过程如下：



在 RLW 中间插入数值的时候，会涉及到部分 Word 的移位，影响性能。所以谷歌官方建议使用者从小到大来插入数据。



官方说明如下：

\* Though you can set the bits in any order (e.g., `set(100)`, `set(10)`, `set(1)`,  
\* you will typically get better performance if you set the bits in increasing order (e.g., `set(1)`, `set(10)`, `set(100)`)  
\* Setting a bit that is larger than any of the current set bit

\* is a constant time operation. Setting a bit that is smaller than an already set bit can require time proportional to the compressed size of the bitmap, as the bitmap may need to be rewritten.

好了，有关 Bitmap 的优化就介绍到这里。感谢各位同学的支持！



## 几点说明：

1. 该项目最初的技术选型并非Mysql，而是内存数据库hana。本文为了便于理解，把最初的存储方案写成了Mysq数据库。